Homework 1

1. ** (2 min.) Write a Haskell function named `largest` that takes in 2 `String` arguments and returns the longer of the two. If they are the same length, return the first argument.

Example:

`largest “cat” “banana”` should return `“banana”`.

`largest “Carey” “rocks”` should return `“Carey”`.

Solution:

```
largest :: String -> String -> String
largest first second =
    if length first >= length second then first else second
```

2. ** (4 min.) Barry Snatchenberg is an aspiring Haskell programmer. He wrote a function named `reflect` that takes in an `Integer` and returns that same `Integer`, but he wrote it in a very funny way:

```
reflect :: Integer -> Integer
reflect 0 = 0
reflect num
  | num < 0 = (-1) + reflect num+1
  | num > 0 = 1 + reflect num-1
```

He finds that when he runs his code, it always causes a stack overflow (infinite recursion) for any non-zero argument! What is wrong with Barry’s code (i.e. can you fix it so that it works properly)?

Barry is missing parentheses around num+1 and num-1. In its current state, the program is functionally equivalent to:

```
reflect :: Integer -> Integer
reflect 0 = 0
reflect num
  | num < 0 = (-1) + (reflect num) + 1
  | num > 0 = 1 + (reflect num) - 1
```

which causes an infinite loop.

To fix the program, we need only to add some parentheses to fix the associativity:

```
reflect :: Integer -> Integer
reflect 0 = 0
reflect num
  | num < 0 = (-1) + reflect (num+1)
  | num > 0 = 1 + reflect (num-1)
```

3. **

a) (2 min.) Write a Haskell function named `all_factors` that takes in an `Integer` argument and returns a list containing, in ascending order, all factors of that integer. You may assume that the argument is always positive. **Your function’s implementation should be a single, one-line list comprehension.**

Example:

`all_factors 1` should return `[1]`.

`all_factors 42` should return `[1, 2, 3, 6, 7, 14, 21, 42]`.

Solution:

```haskell
    all_factors :: Integer -> [Integer]
    all_factors num =
      [x | x <- [1..num], num `mod` x == 0]
```

b) (3 min.) A perfect number is defined as a positive integer that is equal to the sum of its proper divisors (where "proper divisors" refers to all of its positive whole number factors, excluding itself). For example, 6 is a perfect number because its proper divisors are 1, 2 and 3 and 1 + 2 + 3 = 6.

Using the `all_factors` function, write a Haskell expression named `perfect_numbers` whose value is a **list comprehension** that generates an infinite list of all perfect numbers (even though it has not been proved yet whether there are infinitely many perfect numbers 😉).

Example:

`take 4 perfect_numbers` should return `[6, 28, 496, 8128]`.

Hint: You may find the init and sum functions useful.

Solution:

```haskell
    perfect_numbers :: [Integer]
    perfect_numbers =
      [x | x <- [1..], sum (init (all_factors x)) == x]
```

** (4 min.) Write a pair of Haskell functions named `is_odd` and `is_even` that each take in 1 `Integer` argument and return a `Bool` indicating whether the integer is odd or even respectively. You may assume that the argument is always positive.

Example:

`is_even 8` should return `True`.

`is_odd 8` should return `False`.

Hint: The functions can call one another in their implementations. (This is called mutual recursion).

Solutions:

```haskell
    -- with if statements
    is_odd :: Integer -> Bool
    is_odd x =
     if x == 0 then False else is_even (x-1)

    is_even :: Integer -> Bool
    is_even x =
     if x == 0 then True else is_odd (x-1)
```

```haskell
    -- with guards
    is_odd :: Integer -> Bool
    is_odd x
     | x == 0 = False
     | otherwise = is_even (x-1)

    is_even :: Integer -> Bool
    is_even x
```

```
  | x == 0 = True
  | otherwise = is_odd (x-1)


-- with pattern matching
is_odd :: Integer -> Bool
is_odd 0 = False
is_odd x = is_even (x-1)

is_even :: Integer -> Bool
is_even 0 = True
is_even x = is_odd (x-1)
```

4. (6 min.) Write a function named `count_occurrences` that returns the number of ways that all elements of list a_1 appear in list a_2 in the same order (though a_1's items need not necessarily be consecutive in a_2). The empty sequence appears in another sequence of length `n` in 1 way, even if `n` is 0.

Examples:

`count_occurrences [10, 20, 40] [10, 50, 40, 20, 50, 40, 30]` should return 1.

`count_occurrences [10, 40, 30] [10, 50, 40, 20, 50, 40, 30]` should return 2.

`count_occurrences [20, 10, 40] [10, 50, 40, 20, 50, 40, 30]` should return `0`.

`count_occurrences [50, 40, 30] [10, 50, 40, 20, 50, 40, 30]` should return 3.

`count_occurrences [] [10, 50, 40, 20, 50, 40, 30]`
should return 1.

`count_occurrences [] []` should return 1.

`count_occurrences [5] []` should return `0`.

Solution:

```
count_occurrences :: [Integer] -> [Integer] -> Integer
count_occurrences [] _ = 1
count_occurrences _ [] = 0
count_occurrences (x:xs) (y:ys)
  | x == y = count_occurrences xs ys + other_occurrences
  | otherwise = other_occurrences
 where other_occurrences = count_occurrences (x:xs) ys
```

Homework 2

a) (2 min.) Use the `map` function to write a Haskell function named `scale_nums` that takes in a list of `Integers` and an `Integer` named `factor`. It should return a new list where every number in the input list has been multiplied by `factor`.

Example:

`scale_nums [1, 4, 9, 10] 3` should return `[3, 12, 27, 30].`

```
scale_nums :: [Integer] -> Integer -> [Integer]
scale_nums xs factor = map (\x -> x * factor) xs
```

b) (2 min.) Use the `filter` and `all` functions to write a Haskell function named `only_odds` that takes in a list of `Integer` lists, and returns all lists in the input list that only contain odd numbers (in the same order as they appear in the input list). Note that the empty list vacuously satisfies this requirement.

Example:

`only_odds [[1, 2, 3], [3, 5], [], [8, 10], [11]]` should return `[[3, 5], [], [11]]`.

Solution:

```
-- note the partial application with all
only_odds :: [[Integer]] -> [[Integer]]
only_odds xs = filter (all (\x -> mod x 2 /= 0)) xs
```

c) (2 min.) In Homework 1, you wrote a `largest` function that returns the larger of two words, or the first if they are the same length:

```
largest :: String -> String -> String
largest first second =
    if length first >= length second then first else second
```

Use one of `foldl` or `foldr` and the `largest` function to write a Haskell function named `largest_in_list` that takes in a list of `String`s and returns the longest `String` in the list. If the list is empty, return the empty string. If there are multiple strings with the same maximum length, return the one that appears first in the list.

Example:

`largest_in_list ["how", "now", "brown", "cow"]` should return `"brown"`.

`largest_in_list ["cat", "mat", "bat"]` should return `"cat"`.

```
largest_in_list :: [String] -> String
largest_in_list xs = foldl largest "" xs
```

a) (5 min.) Write a Haskell function named `count_if` that takes in a predicate function of type `(a -> Bool)` and a list of type `[a]`. It should return an `Int` representing the number of elements in the list that satisfy the predicate.

Examples:

`count_if (\x -> mod x 2 == 0) [2, 4, 6, 8, 9]` should return `4`.

`count_if (\x -> length x > 2) ["a", "ab", "abc"]` should return `1`.

```
count_if :: (a -> Bool) -> [a] -> Int
count_if predicate [] = 0
count_if predicate (x:xs)
    | (predicate x) = 1 + count_if predicate xs
    | otherwise = count_if predicate xs
```

b) (3 min.) Now, reimplement the same function above (call it `count_if_with_filter`),

```
count_if_with_filter :: (a -> Bool) -> [a] -> Int
count_if_with_filter predicate xs = length (filter predicate xs)
```

c) (3 min.) Now, reimplement the same function above (call it `count_if_with_fold`)

```
count_if_with_fold :: (a -> Bool) -> [a] -> Int
count_if_with_fold predicate xs =
  let count acc x = if predicate x then acc + 1 else acc
```

```haskell
    in foldl count 0 xs
```

a) (3 min.) Explain the difference between currying and partial application.

Currying is the process of taking a function of n arguments and equivalently transforming it into a chain of functions that each only take one argument. Partial application, instead, refers to the process of passing k arguments, where `0 < k < n`, to a curried function that takes n arguments. This yields another function that accepts `n-k` arguments.

b) (4 min.) Suppose we have a Haskell function with type `a -> b -> c`. Consider the other two function types:

i. `(a -> b) -> c`

ii. `a -> (b -> c)`

Is `a -> b -> c` equivalent to i, ii, both, or neither? Why?

It is equivalent to ii. The key is to recognize that Haskell automatically curries functions; if `f :: a -> b -> c`, then invoking `f` with only one argument will yield another function of type `b -> c` (partial application).

Therefore, `a -> b -> c` is not equivalent to i because i describes a function that accepts a single argument (another function of type `a -> b`), whereas
`a -> b -> c` accepts two arguments. However, it is equivalent to ii because ii is simply the curried form of `a -> b -> c` and, as previously discussed, Haskell performs currying automatically.

c) (2 min.) Consider the following Haskell function:

```haskell
foo :: Integer -> Integer -> Integer -> (Integer -> a) -> [a]
foo x y z t = map t [x,x+z..y]
```

Rewrite the implementation of `foo` as a chain of lambda expressions that each take in **one** variable to demonstrate the form of a curried function.

```haskell
foo = \x -> \y -> \z -> \t -> map t [x,x+z..y]
```

1. Consider the following Haskell function:

```haskell
f a b =
  let c = \a -> a   -- (1)
      d = \c -> b   -- (2)
  in \e f -> c d e  -- (3)
```

a) (1 min.) What variables (if any) are captured in the lambda labeled `(1)`?

There are no captured variables. The parameter to the lambda (a in `\a`) "shadows" (hides) the outer a parameter that's passed into `f`. So the outer a is not captured. The line is effectively equivalent to "c = \x -> x" (or any other name for the bound variable).

(1 min.) What variables (if any) are captured in the lambda labeled `(2)`?

b is captured (the second parameter of `f`). c is not captured; the line is effectively equivalent to "d = \y -> b".

b) (1 min.) What variables (if any) are captured in the lambda labeled `(3)`?

c and d are captured (from the `let` declarations). We can understand this as c,d being replaceable with the implementations defined by the `let` declarations. Note that the f in this line has nothing to do with the name of the function being f.

c) (4 min.) Suppose we invoke `f` in the following way:

```
f 4 5 6 7
```

What are the names of the variables that the passed in values (4, 5, 6, and 7) are bound to? Which of the values/variables (if any) are actually referenced in the implementation of `f`? Explain.

This answer is as instructive as we can make it - this level of detail will not be required on the exam.

After `f` is invoked with two arguments, the lambda function `(\e f -> c d e)` is returned, which accepts another two arguments. So, 4 is bound to a, 5 is bound to b, and then the values of 6 and 7 are passed to the returned lambda. 6 is then bound to e, and 7 is bound to `f`. Only the variables bound to 5 and 6 are actually referenced. In lambda `(1)`, the a in the return expression refers to the lambda parameter a, not the function parameter bound to 4. Lambda `(2)` uses the captured variable b, the variable bound to 5. Lambda `(3)` uses e, the variable bound to 6, but not `f`, the one bound to 7.

What's going on with `\e f -> c d e`? Both c and d are lambda functions, so are we passing d as the parameter to c? Yes! Function c (`c = \a -> a`) takes one parameter, a, and returns the value of that same parameter. In this case, the argument to c is the function d, in the lambda `\e f -> c d e`. So, the call `c d e` passes d as an argument to c, which just re-returns d. This results in the remaining expression of `d e`. Haskell then calls lambda d (`d = \c -> b`) passing e as the argument. The d function finally just returns the captured value of b, which is 5 (without using the value of e at all!). This is a tricky question, but hopefully it got your gears turning.

Alternative explanation:

Calling "f 4 5 6 7" binds in order: a=4, b=5, e=5, f=7. Replacing each line with what we worked out from parts a-c, we have:

```
f a b =
  let c = \x -> x    -- (1)
      d = \y -> b    -- (2)
  in \e z -> c d e  -- (3)
```

Evidently, a,f are unused since they are not referenced in the function body.

c is the identity function, d is a function that returns 5 (the value of b) for all inputs.

Evaluating line 3 gives us that we don't care about the 2nd parameter of the lambda and that (c d) = d, \e -> d e =5.

Thus, b and e are referenced but only b actually affects the outcome of the function.

2. (5 min.) C allows you to point to functions, as in the following code snippet:

```
int add(int a, int b) {
  return a + b;
}

int main() {
  // Declare a pointer named `fptr` that points to a function
  // that takes in two int arguments and returns another int
  int (*fptr)(int, int);

  // Assign the address of the `add` function to `fptr`
  fptr = &add;

  // Invoke the function using `fptr`. This returns 8.
```

```
    (*fptr)(3, 5);
}
```

Function pointers are first-class citizens like any other pointer in C: they can be passed as arguments, used as return values, and assigned to variables. As a reminder, however, C does not support nested functions.

Compare function pointers in C with closures in Haskell. Are Haskell closures also first-class citizens? What (if any) capabilities do function pointers have that closures do not (and vice versa)?

Closures are also first-class citizens in Haskell. Lambda expressions and nested functions that capture variables can be passed, returned, and assigned. However, Haskell closures can capture local variables (and extend the lifetime of the captured variables in memory) whereas C function pointers cannot. Function pointers are merely addresses to executable code; they have no notion of an environment. As such, you cannot capture variables allocated on the stack - which include function parameters and local variables (but you can in Haskell, due to its use of garbage collection - it keeps captured variables around as long as necessary).

3. Haskell allows you to define your own custom data types. In this question, you'll look at code examples and use them to write your own (that is distinct from the ones shown).

Consider the following code example:

```haskell
data Triforce = Power | Courage | Wisdom

wielder :: Triforce -> String
wielder Power = "Ganon"
wielder Courage = "Link"
wielder Wisdom = "Zelda"

princess = wielder Wisdom
```

a) (2 min.) Define a new Haskell type `InstagramUser` that has two value constructors (without parameters) - `Influencer` and `Normie`.

```haskell
data InstagramUser = Influencer | Normie
```

b) (2 min.) Write a function named `lit_collab` that takes in two `InstagramUsers` and returns `True` if they are both `Influencers` and `False` otherwise.

```haskell
lit_collab :: InstagramUser -> InstagramUser -> Bool
lit_collab Influencer Influencer = True
lit_collab _ _ = False
```

Consider the following code example:

```haskell
data Character = Hylian Int | Goron | Rito Double | Gerudo | Zora

describe :: Character -> String
describe (Hylian age) = "A Hylian of age " ++ show age
describe Goron = "A Goron miner"
describe (Rito wingspan) = "A Rito with a wingspan of " ++ show wingspan
++ "m"
describe Gerudo = "A mighty Gerudo warrior"
describe Zora = "A Zora fisher"
```

c) (2 min.) Modify your `InstagramUser` type so that the `Influencer` value constructor takes in a list of Strings representing their sponsorships.

```
data InstagramUser = Influencer [String] | Normie
```

d) (3 min.) Write a function `is_sponsor` that takes in an `InstagramUser` and a `String` representing a sponsor, then returns `True` if the user is sponsored by `sponsor` (this function always returns `False` for `Normies`).

```
is_sponsor :: InstagramUser -> String -> Bool
is_sponsor Normie _ = False
is_sponsor (Influencer sponsors) sponsor =
  sponsor `elem` sponsors
```

Consider the following code example:

```
data Quest = Subquest Quest | FinalBoss

count_subquests :: Quest -> Integer
count_subquests FinalBoss = 0
count_subquests (Subquest quest) = 1 + count_subquests quest
```

e) (2 min.) Modify your `InstagramUser` type so that the `Influencer` value constructor also takes in a list of other `InstagramUsers` representing their followers (after their sponsors).

```
data InstagramUser = Influencer [String] [InstagramUser] | Normie
```

f) (3 min.) Write a function `count_influencers` that takes in an `InstagramUser` and returns an `Integer` representing the number of `Influencers` that are following that user (this function always returns `0` for `Normies`).

```
-- foldl-based solution
count_influencers :: InstagramUser -> Integer
count_influencers Normie = 0
count_influencers (Influencer _ followers) =
  let count acc (Influencer _ _) = acc + 1
      count acc Normie = acc
  in foldl count 0 followers




-- hand-coded recursive solution
count_influencers:: InstagramUser -> Integer
count_influencers Normie = 0
count_influencers (Influencer _ followers) =
  let count ((Influencer s f):xs) = 1 + count xs
      count (_:xs) = count xs
      count [] = 0
  in count followers
```

g) (2 min.) Use GHCi to determine the type of `Influencer` using the command `:t Influencer`. What can you infer about the type of custom value constructors?

Value constructors are just functions that return an instance of the custom data type!

a) (5 min.) Using C++, write a function named `longestRun` that takes in a `vector` of booleans and returns the length of the longest consecutive sequence of `true` values in that vector.

Examples:

Given `{true, true, false, true, true, true, false}`,

`longestRun(vec)` should return `3`.

Given `{true, false, true, true}`, `longestRun(vec)` should return `2`.

```
int longestRun(vector<bool> vec) {
  int currentMax = 0;
  int currentRun = 0;
  for (bool element : vec) {
    if (element) currentRun++;
    else {
      currentMax = max(currentMax, currentRun);
      currentRun = 0;
    }
  }
  return max(currentMax, currentRun);
}
```

b) ** (10 min.) Using Haskell, write a function named `longest_run` that takes in a list of `Bool`s and returns the length of the longest consecutive sequence of `True` values in that list.

Examples:

`longest_run [True, True, False, True, True, True, False]` should return `3`.

`longest_run [True, False, True, True]` should return `2`.

Solution using helper parameters:

```
longest_run :: [Bool] -> Int
longest_run xs =
  let
    longest_run_helper [] current_run current_max =
      max current_run current_max
    longest_run_helper (x:xs) current_run current_max =
      if x
        then longest_run_helper xs (current_run+1) current_max
        else longest_run_helper xs 0 (max current_run current_max)
  in
    longest_run_helper xs 0 0
```

Fancy solutions using scanl courtesy of Timothy Gu:

```
longest_run :: [Bool] -> Int
longest_run l = maximum (scanl (\p x -> if x then p + 1 else 0) 0 l)
longest_run :: [Bool] -> Int
longest_run = maximum . scanl (\p x -> if x then p + 1 else 0) 0
```

c) (10 min.) Consider the following C++ class:

```
#import <vector>
using namespace std;

class Tree {
public:
  unsigned value;
  vector<Tree *> children;

  Tree(unsigned value, vector<Tree *> children) {
    this->value = value;
    this->children = children;
  }
};
```

Using C++, write a function named `maxTreeValue` that takes in a `Tree` pointer `root` and returns the largest value within the tree. If `root` is `nullptr`, return `0`. **This function may not contain any recursive calls.**

```
#import <queue>

unsigned maxValue(Tree *root) {
  unsigned currentMax = 0;
  queue<Tree *> nodesToExplore;
  nodesToExplore.push(root);

  while (!nodesToExplore.empty()) {
    Tree *current = nodesToExplore.front();
    nodesToExplore.pop();
    if (!current) continue;

    unsigned value = current->value;
    vector<Tree *> children = current->children;

    if (value > currentMax)
      currentMax = value;
    for (Tree *child : children)
      nodesToExplore.push(child);
  }

  return currentMax;
}
```

** (5 min.) Consider the following Haskell data type:

```
data Tree = Empty | Node Integer [Tree]
```

Using Haskell, write a function named `max_tree_value` that takes in a `Tree` and returns the largest `Integer` in the `Tree`. Assume that all values in the tree are non-negative. If the root is `Empty`, return `0`.

Example:

`max_tree_value (Node 3 [(Node 2 [Node 7 []]), (Node 5 [Node 4 []])])` should return 7.

Solutions:

```
-- hand-coded recursive solution
max_tree_value :: Tree -> Integer
max_tree_value Empty = 0
max_tree_value (Node val []) = val
max_tree_value (Node val lst) = max val (max_aux lst)
  where
    max_aux [] = 0
    max_aux (x:xs) = max (max_tree_value x) (max_aux xs)
-- map-based solution
max_tree_value :: Tree -> Integer
max_tree_value Empty = 0
max_tree_value (Node value children) =
  case children of
    [] -> value
    xs -> max value (maximum (map max_tree_value xs))
```

9. (10 min.) Write a Haskell function named `fibonacci` that takes in an `Int` argument n. It should return the first n numbers of the Fibonacci sequence (for this problem, we'll say that the first two numbers of the sequence are 1, 1).

Examples:

`fibonacci 10` should return `[1,1,2,3,5,8,13,21,34,55].`

`fibonacci -1` should return `[].`

```
-- solution where we build the list in reverse
fibonacci :: Int -> [Integer]
fibonacci n =
  let fib_rev 1 = [1]
      fib_rev 2 = [1, 1]
      fib_rev n =
        let prev_fib_rev = fib_rev (n-1)
            first = head prev_fib_rev
            second = head (tail prev_fib_rev)
        in (first + second) : prev_fib_rev
  in reverse (fib_rev n)



-- less-efficient solution building the list forward
-- second_last is O(n)
fibonacci 1 = [1]
fibonacci 2 = [1, 1]
fibonacci n =
    let second_last xs
          | length xs == 2 = head xs
          | otherwise = second_last (tail xs)
        prev_fib = fibonacci (n-1)
    in prev_fib ++ [last prev_fib + second_last prev_fib]


-- fancy solution using list comprehension that generates an
-- infinite list.
-- this works because of Haskell's lazy evaluation.
fibonacci n =
  let fib = 1 : 1 : [a+b | (a,b) <- zip fib (tail fib)]
  in take n fib
```

Homework 3

1. ** Consider the following Haskell data type:

```haskell
data LinkedList = EmptyList | ListNode Integer LinkedList
  deriving Show
```

a. ** (3 min.) Write a function named `ll_contains` that takes in a `LinkedList` and an `Integer` and returns a `Bool` indicating whether or not the list contains that value.

Examples:

`ll_contains (ListNode 3 (ListNode 6 EmptyList)) 3`

should return `True`.

```haskell
ll_contains :: LinkedList -> Integer -> Bool
ll_contains EmptyList _ = False
ll_contains (ListNode x next) y =
  if x == y then True else ll_contains next y
```

b. ** (3 min.) We want to write a function named `ll_insert` that inserts a value at a given zero-based index into an existing `LinkedList`. Provide a type definition for this function, explaining what each parameter is and justifying the return type you chose.

```haskell
ll_insert :: LinkedList -> Integer -> Integer -> LinkedList
```

Given the definition of the `LinkedList` data type, there is no way for us to modify a `LinkedList` passed to `ll_insert`. Therefore, what makes most sense is for `ll_insert` to return a new `LinkedList` where the value has been inserted at the desired index. The first parameter of our function will be the `LinkedList` to insert into, the second the value, and the third the index.

c. ** (5 min.) Implement the `ll_insert` function. If the insertion index is 0 or negative, insert the value at the beginning. If it exceeds the length of the list, insert the value at the end. Otherwise, the value should have the passed-in insertion index after the function is invoked.

```haskell
ll_insert :: LinkedList -> Integer -> Integer -> LinkedList
ll_insert EmptyList x _ = ListNode x EmptyList
ll_insert (ListNode y rest) x index
  | index <= 0 = ListNode x (ListNode y rest)
  | otherwise = ListNode y (ll_insert rest x (index-1))
```

** (8 min.) Consider the following Python scripts:

```python
class Box:
  def __init__(self, value):
    self.value = value

def quintuple(num):
  num *= 5

def box_quintuple(box):
  box.value *= 5

num = 3
box = Box(3)
```

```
quintuple(num)
box_quintuple(box)
print(f"{num} {box.value}")
```

Running this script yields the output:

```
3 15
```

Explain, in detail, why `box`'s `value` attribute was mutated but `num` was not. Your answer should explicitly mention Python's parameter passing semantics.

This answer is as instructive as we can make it - this level of detail will not be required on the exam.

In Python, everything is an object; every variable is essentially a pointer to an object. This is true for primitive data types too (including `int`, `bool` and `str`). Furthermore, Python uses pass by object reference for function parameters: if we have a function `f` with parameter `x`, and we invoke `f` by passing some variable `a`, then `x` merely points to whatever object that `a` points to. So, for example, when we refer to `num` *inside* the `quintuple` function, it is a separate pointer to the original object that the `num` *outside* the function points to. In other words, we have two different pointers that both point to the object 3.

Next, it is important to note that when you reassign a variable to a new value, you aren't changing the value of the old object pointed to by that variable - you make the variable point to the new object you're assigning it. In the case of the expression `num *= 5` (which is syntactic sugar for `num = num * 5`), we change the num variable *inside* of the `quintuple` function such that it now points at the object 15. We do not modify the object that it previously pointed to (the `int` 3), nor do we change what the num variable *outside* of the function points to (which is why 3 is still printed).

However, custom classes in Python (like the Box class) *do* store their attributes separately (as alluded to in the hint, they're stored in a dictionary object named `__dict__`). So the expression `box.value *= 5` modifies the `value` attribute on the same object pointed at by both `box` variables (i.e. the one inside the function *and* outside. Both variables point to the same object). This is why 15 is printed.

d. ** Here's our second script, taken from a previous CS131 midterm exam:

```
class Comedian:
  def __init__(self, joke):
    self.__joke = joke

  def change_joke(self, joke):
    self.__joke = joke

  def get_joke(self):
    return self.__joke

def process(c):
 # Line A
 c[1] = Comedian("joke3")
 c.append(Comedian("joke4"))
 c = c + [Comedian("joke5")]
 c[0].change_joke("joke6")
```

```python
def main():
 c1 = Comedian("joke1")
 c2 = Comedian("joke2")
 com = [c1,c2]
 process(com)
 c1 = Comedian("joke7")
 for c in com:
  print(c.get_joke())
```

i. Assuming we run the main function, what will this program print out?

 `Answers for part i and ii are below.`

ii. Assuming we removed the comment on line A and replaced it with:

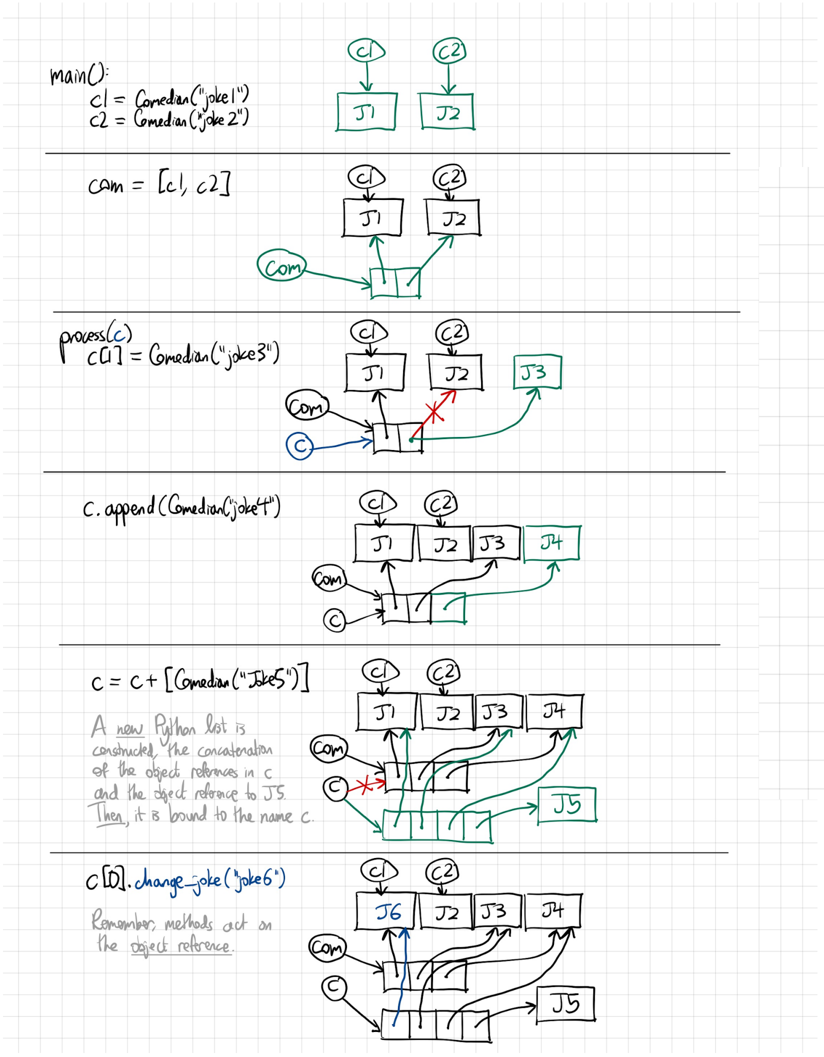
```
      c = copy.copy(c)          # initiate a shallow copy
```

If we run the main function, what would the program print?

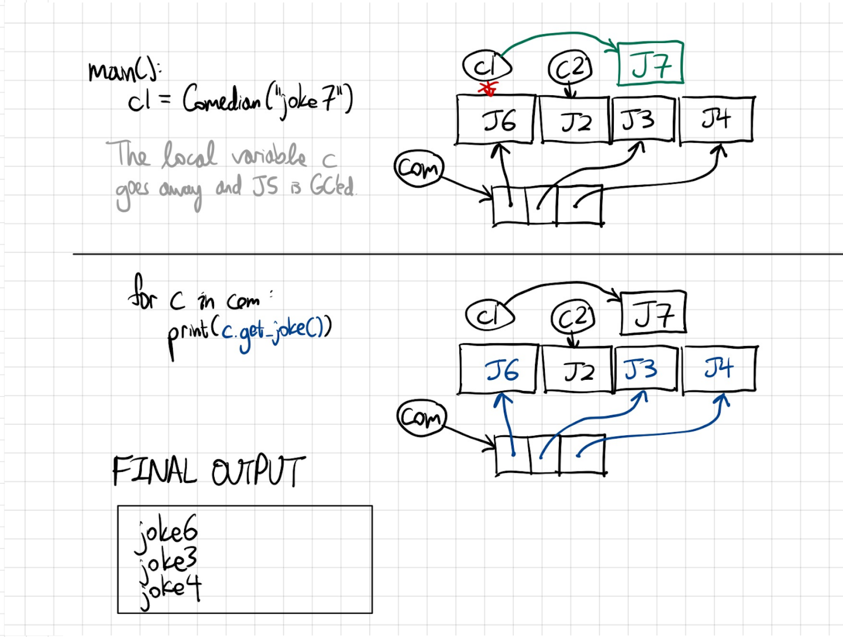
`Answers for part i and ii, as written by our former student Vincent Lin:`

- Green describes a new allocation, extension, etc. as a result of the current line.
- Red describes something that has been removed as a result of the current line.
- Blue describes what's being referenced in a method call on the current line.

I also included some notes in gray describing non-obvious behavior that Python performs behind the scenes.

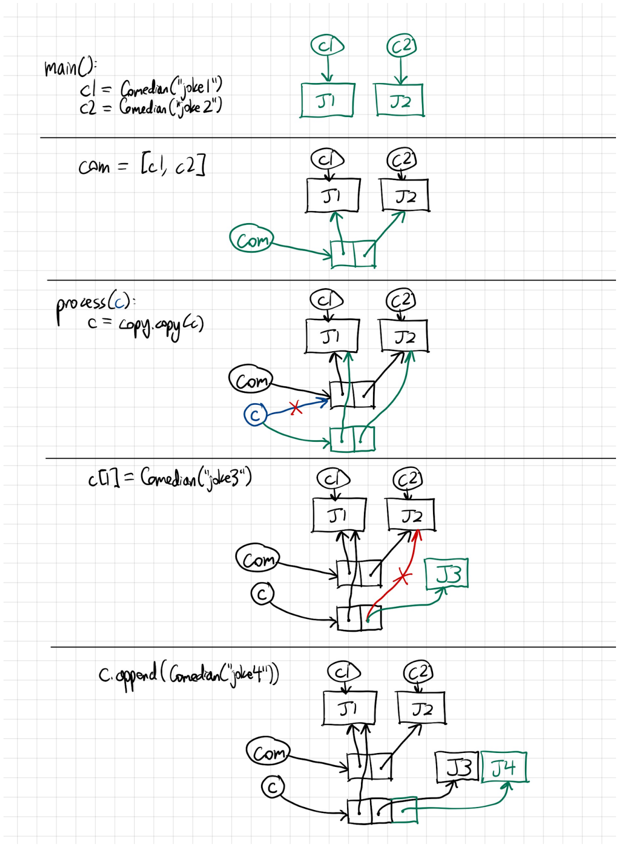


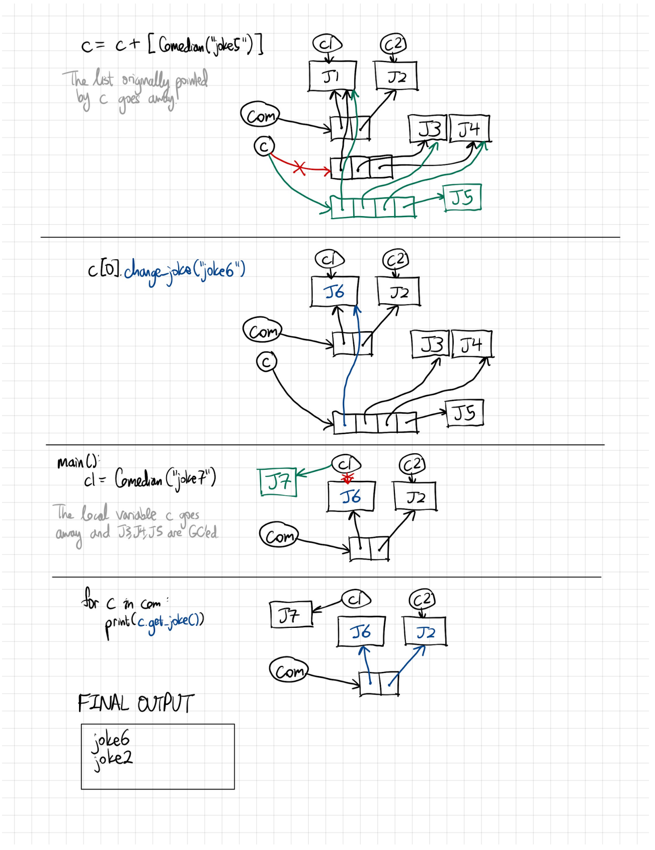
main():
c1 = Comedian("joke1")
c2 = Comedian("joke2")
com = [c1, c2]
process(c)
c[1] = Comedian("joke3")
c.append(Comedian("joke4"))
c = c + [Comedian("Joke5")]
A new Python list is constructed, the concatenation of the object references in c and the object reference to J5. Then, it is bound to the name c.
c[0].change_joke("joke6")
Remember, methods act on the object reference.
c1
c2
com
c
J1
J2
J3
J4
J5
J6



```
joke6
joke3
joke4
```

Part (b)




c = c + [Comedian("joke5")]
The list originally pointed by c goes away!
c[0].change_joke("joke6")
main():
c1 = Comedian("joke7")
The local variable c goes away and J3,J4,J5 are GC'ed
for c in com:
print(c.get_joke())
FINAL OUTPUT
joke6
joke2
c1
c2
J1
J2
J3
J4
J5
J6
J7
com
c

2.

3. ** (5 min.) Consider the following output from the Python 3 REPL:

```
>>> class Foo:
...   def __len__(self):
...     return 10
...
>>> len(Foo())
10
>>> len("blah")
4
>>> len(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
```

Explain why Python allows you to pass an object of type `Foo` or `str` to `len`, but not one of type `int`. Your answer should explicitly reference Python's typing system.

Python is duck-typed, which means that the type of an object is less important than the attributes and methods it defines. For example, functions in Python do not discriminate by requiring you to pass objects of a certain type - you can pass in anything you'd like. However, if you try to invoke a method or reference an attribute that does not exist on an object, Python will raise an error.

Built-in functions work the same way. In the example, you can see that for the `len` function to return without throwing an exception, the caller merely needs to pass in an object that has the `__len__` function defined. In other words, the

implementation of `len("blah")` at some point calls `"blah".__len__()`. `int` objects do not have a `__len__()` function, so the built-in `len` raises a `TypeError`.

4. Consider the following `Duck` class, and two functions that check whether or not an object is a `Duck`:

```
class Duck:
  def __init__(self):
    pass # Empty initializer
```

```
def is_duck_a(duck):
  try:
    duck.quack()
    return True
  except:
    return False
```

```
def is_duck_b(duck):
  return isinstance(duck, Duck)
```

a. (2 min.) Write a **new** class such that if we passed an instance of it to both functions, `is_duck_a` would return `True` but `is_duck_b` would return `False`.

Solution:

```
class RubberDuck:
  def quack(self):
    print("Squeak")
```

(2 min.) Write a **new** class such that if we passed an instance of it to both functions, `is_duck_a` would return `False` but `is_duck_b` would return `True`.

Solution:

```
class Mallard(Duck):
  pass # empty class, no new added methods/fields
```

b. (5 min.) Which function is more Pythonic: `is_duck_a` or `is_duck_b`? This reference may help provide some insight.

`is_duck_a` is more Pythonic. In fact, both approaches have canonical names: "Easier to ask for forgiveness than permission" (EAFP) and "Look before you leap" (LBYL) respectively. Essentially, in Python it's considered more robust to make assumptions (e.g. that a `Duck` has a `quack()` method) and handle errors when those assumptions do not hold than to assert conditions ahead of time (am I dealing with a `Duck` object?) and carry on assuming they will always be true. For a concrete example of why, check out this old email from the Python mailing list.

5. ** (3 min.) Consider the following function that takes in a list of integers and another integer `k` and returns the largest sum you can obtain by summing `k` consecutive elements in the list. Fill in the blanks so the function works correctly.

`largest_sum([3,5,6,2,3,4,5], 3)` should return 14.

```
def largest_sum(nums, k):
  if k < 0 or k > len(nums):
    raise ValueError
  elif k == 0:
    return 0

  max_sum = None
  for i in range(len(nums)-k+1):
    sum = 0
    for num in nums[i:i+k]:
      sum += num
    if max_sum is None or sum > max_sum:
      max_sum = sum
  return max_sum
```

a. (3 min.) The function in part a) runs in `O(nk)` time where `n` is the length of `nums`, but we can use the **sliding window technique** to improve the runtime to `O(n)`.

Imagine we know the sum of `k` consecutive elements starting at index `i`. We are interested in the next sum of `k` consecutive elements starting at index `i+1`. Notice that the only difference between these two sums is the element at index `i` (which becomes excluded) and the element at index `i+k` (which becomes included).

We can compute each next sum by subtracting the number that moved out of our sliding window and adding the one that moved in. Fill in the blanks so the function works correctly.

```
def largest_sum(nums, k):
  if k < 0 or k > len(nums):
    raise ValueError
  elif k == 0:
    return 0

  sum = 0
  for num in nums[0:k]:
    sum += num

  max_sum = sum
  for i in range(0, len(nums)-k):
    sum -= nums[i]
    sum += nums[i+k]
    max_sum = max(sum, max_sum)
  return max_sum
```

** We're going to design an event-scheduling tool, like a calendar. Events have a `start_time` and an `end_time`. For simplicity, we will model points in time using `int`s that represent the amount of seconds past some reference time (normally, we would use the Unix epoch, which is January 1, 1970).

b. ** (3 min.) Design a Python class named `Event` which implements the above functionality. Include an initializer that takes in a `start_time` and `end_time`. If `start_time >= end_time`, raise a `ValueError`.

Solution:

```
class Event:
  def __init__(self, start_time, end_time):
    if start_time >= end_time:
      raise ValueError
```

```
        self.start_time = start_time
        self.end_time = end_time
```

c. ** (3 min.) Write a Python class named `Calendar` that maintains a private list of scheduled events named `__events`.

Solution:

```
class Calendar:
  def __init__(self):
    self._events = []

  def get_events(self):
    return self._events

  def add_event(self, event):
    if not isinstance(event, Event):
      raise TypeError
    self._events.append(event)
```

Notice that we used `isinstance` even though, in the solution to question 6, we argued that it is more Pythonic to ask for forgiveness than to look before you leap. This would be the only way to check if an object is truly of a particular type. An alternative approach we could have taken would be to require that any object passed in implements the same set of methods as the Event class, rather than require only Event (or subclasses of Event) objects be passed in. This would enable duck typing to be used.

d. ** (3 min.) Consider the following subclass of `Calendar`:

```
class AdventCalendar(Calendar):
  def __init__(self, year):
    self.year = year

advent_calendar = AdventCalendar(2022)
print(advent_calendar.get_events())
```

Running this code as-is (assuming you have a correct `Calendar` implementation) yields the following output:

```
AttributeError: 'AdventCalendar' object has no attribute '__events'. Did you mean:
'get_events'?
```

Explain why this happens, and list two different ways you could fix the code **within the class** so the snippet instead prints:

```
[]
```

While the `AdventCalendar` class does inherit from the `Calendar` class, it does not explicitly invoke its base class' initializer via `super().__init__`. Because of this, `Calendar`'s initializer (which adds an empty `events` list to the instance) is never run. Notice, however, that we do still inherit the methods defined on `Calendar` (namely `get_events`).

To fix this, inside `AdventCalendar`'s `__init__` function we can either add an explicit call to the super initializer:

```
super().__init__()
```

or we can just add an `events` attribute ourselves. Of course, calling the superclass initializer would be better practice (since it leads to less repetitive code, especially in cases where the base class initializer has additional logic).

```
self._events = []
```

6. Suppose we have 2 languages. The first is called I-Lang and is an interpreted language. The interpreter receives lines of I-Lang and efficiently executes them line by line. The second is called C-Lang and is a compiled language. Assume that it takes the same time to write an I-Lang script as a C-Lang program if both perform the same function.

a. (2 min.) Suppose we have an I-Lang script and a C-Lang executable that functionally perform the exact same thing. If we execute both at the same time, which do you expect to be faster and why?

We would expect the C-Lang executable to be faster. The C-Lang program is compiled directly to machine code, whereas the interpreted I-Lang code is translated into lower-level representations at run-time. This extra step creates additional overhead which will inevitably cause it to be slower than the pre-compiled code

b. (2 min.) Suppose Jamie and Tim are two equally competent students developing a web server that sends back a simple, plaintext HTML page. Jamie writes her server in I-Lang, whereas Tim writes his in C-Lang. Assuming that the server will be deployed locally, who will most likely have the server running first, and why?

We would expect Jamie to get the web server running first. From start to finish, the only difference between both development processes is that Jamie needs to execute her code using the interpreter, whereas Tim needs to compile his into an executable. In general, the compilation process takes a nontrivial amount of time, whereas the I-Lang interpreter can begin executing lines of code right away.

It could also be correct to argue that Tim would be able to get the server running first. If the I-Lang interpreter was extremely slow, or if the C-Lang compiler was abnormally fast, then as in part a), the additional overhead associated with interpreting each line of I-Lang may be enough to make Tim the winner. For most modern languages, however, compilation (which involves preprocessing, compiling, assembling, and linking) is a process that takes much longer than simply interpreting lines of code.

c. (2 min.) Jamie and Tim have a socialite friend named Connie who uses a fancy new SmackBook Pro. The SmackBook Pro has a special kind of chip called the N1, which has a proprietary machine language instruction set (ISA) completely unique to anything else out there. If you are familiar with Rosetta, assume the SmackBook Pro has **no** built-in emulator/app translator. Jamie and Tim have less-fancy computers with Intel chips.

Connie has a native copy of the I-Lang interpreter on her computer. Jamie sends Connie her web server script, and Tim sends Connie his pre-compiled executable. Will Connie be able to execute Jamie's script? What about Tim's executable?

Connie will be able to execute Jamie's script because she already has a working version of the I-Lang interpreter on her computer. However, she will not be able to run Tim's executable, which has already been compiled into machine code that is tailored for Intel's ISA - it isn't portable.

7. (10 min.) Consider the following code snippet that compares the performance of matrix multiplication using a hand-coded Python implementation versus using numpy with output:

```
Hand-coded implementation took 0.043227195739746094 seconds
numpy took 0.0004067420959472656 seconds
```

Assuming that the implementations of `dot_product` and `matrix_multiply` are reasonably optimized, provide an explanation for this discrepancy in performance.

Solution:

The key to answering this question is to recognize that under the hood (as referenced by the hint), numpy's implementation (which includes functions like `dot`) heavily relies on pre-compiled, optimized C code. There are therefore two main reasons (only the first of which we would expect you to reasonably elaborate on on an exam) that numpy is significantly faster.

Firstly, as discussed in previous questions, compiled code is much more performant than interpreted code, because the latter involves the overhead of translation to lower-level representations at runtime. This is especially pertinent for our code, which will call the `dot_product` function at least 1000 times. For each invocation, the Python interpreter will convert the bytecode corresponding to `dot_product` into machine-executable instructions. This process has already been done ahead of time for the C code.

Secondly, the underlying memory representation for numpy arrays is much more efficient than that of Python lists. Recall that in Python, everything is an object, which means that the matrix variable actually contains one million pointers. Not only is this less space efficient, there is the chance of poor spatial locality: the objects pointed to in the matrix might be scattered throughout RAM. However, numpy provides support for contiguous arrays where each element is actually next to each other in memory (as in C). This enables more efficient caching techniques at the CPU level and chip-specific optimizations such as SIMD.

8. (5 min.) Consider the following code snippet that uses both class variables and instance variables:

```
class Joker:
 joke = "I dressed as a UDP packet at the party. Nobody got it."

 def change_joke(self):
  print(f'self.joke = {self.joke}')
  print(f'Joker.joke = {Joker.joke}')
  Joker.joke = "How does an OOP coder get wealthy? Inheritance."
  self.joke = "Why do Java coders wear glasses? They can't C#."
  print(f'self.joke = {self.joke}')
  print(f'Joker.joke = {Joker.joke}')

j = Joker()
print(f'j.joke = {j.joke}')
print(f'Joker.joke = {Joker.joke}')
j.change_joke()
print(f'j.joke = {j.joke}')
print(f'Joker.joke = {Joker.joke}')
```

a. What do you expect this program to print out?

Solution:

#note: newlines added for clarity

j.joke = I dressed as a UDP packet at the party. Nobody got it.

Joker.joke = I dressed as a UDP packet at the party. Nobody got it.

self.joke = I dressed as a UDP packet at the party. Nobody got it.

Joker.joke = I dressed as a UDP packet at the party. Nobody got it.

self.joke = Why do Java coders wear glasses? They can't C#.

Joker.joke = How does an OOP coder get wealthy? Inheritance.

j.joke = Why do Java coders wear glasses? They can't C#.

Joker.joke = How does an OOP coder get wealthy? Inheritance.

b. What does each print statement actually print out? Why?

Solution:

The correct answer is above, so this part will just be an explanation of the above.

Note that modifying a class variable on the class namespace, modifies all the instances of the class.

Also, for the field `self.joke`, if there is no instance variable `joke` in the `self` instance, then `self.joke` will reference the class variable `joke` on the class that `self` is an instance of (in this case, Joker). However, if an instance variable `self.joke` exists, it will shadow the class variable of the same name, overriding and hiding it.

```
self.joke = "Why do Java coders wear glasses? They can't C#."
```

creates a new `joke` variable that gets added to the instance of the Joker class.

So, the first 4 lines are the same, since `self.joke` and `Joker.joke` refer to the same class variable.

Then, line 5 refers to the (newly created!) instance variable, and line 6 refers to the changed class variable.

Finally, line 7 refers to the instance variable and line 8 refers to the class variable, so the same situation as lines 5 and 6.

Homework 4

1. ** (3 min.) Consider the following function that takes in a list of integers (either 0 or 1) representing a binary number and returns the decimal value of that number. Fill in the blanks in the list comprehension so the function works correctly.

`convert_to_decimal([1, 0, 1, 1, 0])` should return 22.

```
from functools import reduce
def convert_to_decimal(bits):
  exponents = range(len(bits)-1, -1, -1)
  nums = [bit * 2**x for bit, x in zip(bits, exponents)]
  return reduce(lambda acc, num: acc + num, nums)
```

a) (5 min.) Write a function named `parse_csv` that takes in a list of strings named `lines`. Each string in `lines` contains a word, followed by a comma, followed by some number (e.g. `"apple,8"`).

`parse_csv(["apple,8", "pear,24", "gooseberry,-2"]` should return `[("apple", 8), ("pear", 24), ("gooseberry", -2)]`.

Solution:

```
def parse_csv(lines):
  return [(x, int(y)) for x, y in [line.split(",") for line in lines]]
```

b) (2 min.) Write a function named `unique_characters` that takes in a string `sentence` and returns a set of every unique character in that string.

`unique_characters("happy")` should return `{"h", "a", "p", "y"}`.

```
def unique_characters(sentence):
  return {s for s in sentence}
```

c) (2 min.) Write a function named `squares_dict` that takes in an integer `lower_bound` and an integer `upper_bound` and returns a dictionary of all integers between `lower_bound` and `upper_bound` (inclusive) mapped to their squared value. `squares_dict(1, 5)` should return `{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}`.

```
def squares_dict(lower_bound, upper_bound):
  return {x: x**2 for x in range(lower_bound, upper_bound+1)}
```

2. ** (4 min.) Consider the following function that takes in a string `sentence` and a set of characters `chars_to_remove` and returns `sentence` with all characters in `chars_to_remove` removed. Fill in the blank so the function works correctly. `strip_characters("Hello, world!", {"o", "h", "l"})` should return
`"He, wrd!"`

```
def strip_characters(sentence, chars_to_remove):
  return "".join(filter(lambda x: x not in chars_to_remove, sentence))
```

3. ** (4 min.) Show, by example, whether or not Python supports **closures**. Briefly explain your reasoning.

We need to show that it is possible to bundle together function code with references to outside state (captured variables). We can use either a nested function or lambda for this. The following snippet from the REPL suffices:

```
>>> def foo():
...   b = 5
...   f = lambda x: x + b
...   return f
...
>>> foo()(0)
5
```

`foo` returns a closure that has captured b.

Zeta Programming

4. ** You're working on a project at Zeta to add Haskell-like type inference to Python. (Fun fact: Meta actually does have an open-source gradual typing/type inference Python system called Pyre. Microsoft has one called Pyright)

a) ** (2 min.) Consider this function: if you could add a Haskell-like type annotation, what would it look like? Alternatively, if you can't annotate it like Haskell, why not?

```
from math import sqrt

def nth_fibonacci(n):
  phi = (1 + sqrt(5))/2
  psi = (1 - sqrt(5))/2
  return (phi**n - psi**n)/sqrt(5)
```

For this one, we can annotate it. On an exam, we would expect an annotation of the form:

`nth_fibonacci :: Float -> Float`

How did we get there? In detail:

- phi and psi are both constants; in particular, a `Float`
- the ** operator (exponent) on a `Float` expects another `Float` as its right-hand side argument, so we can infer that n is a `Float`
- a `Float` divided by another `Float` returns a `Float`, so we conclude that the return value is a `Float`

In fact, this function adheres to all FP principles, and we could write an equivalent function in Haskell. In GHCi:

```
nth_fibonacci n =
  let phi = (1 + sqrt(5))/2
      psi = (1 - sqrt(5))/2
  in (phi**n - psi**n)/sqrt(5)
```

`:t nth_fibonacci`
would give us something of the form
`nth_fibonacci :: Float -> Float`

(there's some small nuance about ** only applying to `Float`s in Haskell, but we're not expecting you to know that)

Aside: in a purely mathematical sense, this actually will *always* return an integer for an integer input. But, that involves a math proof, and Haskell's type system doesn't do that. `Integer -> Integer` would also be too narrow of a type!

b) (2 min.) Consider this function. Assume that `network_type` is always a String. Your boss says that it's not possible to annotate this function, because its return type is not the same for every input (which breaks Haskell's rules about functions). You're convinced there's still a way to do this. What could that look like?

```
def get_network_type(obj):
  if not hasattr(obj, 'network_type'):
    return None
  return obj.network_type
```

First, to clarify: this function takes in *any* Python object. If it has the `network_type` attribute, it returns the value of that object; if not, it returns None. If you're curious, read the `hasattr()` docs.

In Python, everything is an object! This means that our input can be *literally anything*: an integer literal, a function, a class we define, or something else. That means we don't put any restriction on what it can be; so, we'll use a type variable for it; we'll call ours `object`.

If we add some extra code, we could write an annotation for the return value. If we have an ADT of the form:

```
data StringOrNone = Str String | None
```

We could then annotate the function like so: `get_network_type :: object -> StringOrNone`

c) ****** (3 min.) Consider this function.

```
def add(a,b):
  return a + b
```

It seems easy, right? Your boss provides the annotation:

```
add :: Num -> Num -> Num
```

Is this the best annotation? Why or why not?

No! In Python, + is overloaded and also operates on other types: lists, strings, and any class that implements the __add()__ method – broader than Num!

(many students answered something to the form of "we want an `int + int` to be an `int`". That is true, though in Haskell, we typically look for the most generic type – this is not how + is typed, for example)

Union Busting

5. ******

a) ****** (4 min.) Examine the following C++ code:

```
union WeirdNumber {
  int n;
  float f;
};

int main() {
  WeirdNumber w;
  w.n = 123;
  std::cout << w.n << std::endl;
  std::cout << w.f << std::endl;
}
```

This evaluates to:

```
123
1.7236e-43
```

Why? What does this say about C++'s type system?

This happens because C++ lets you access any type of the union at any time, even if it's not the one that's "active". `1.7236e-43` happens to be the floating-point representation encoded by the *bits* that 123 sets for an `int`.

In programming language terminology, this is called an untagged union.

Implicitly, this is an argument that C++ is weakly-typed.

- **`float`'s don't have a singular bit representation in C++: their precision depends on compiler flags, machine architecture, etc.!**
  - **We could even interpret this as an implicit cast**
- **Thus, this program does not have clear behavior across different environments; we could frame this as undefined behavior or as type-unsafe behavior.**
  - **In Carey's slides, this is probably in the "unsafe memory access" category, or an undefined type conversion**
- **There is also a more strict definition of "undefined behavior" in the C++ spec. Whether or not this is actually undefined behavior is not clear-cut (see: this SO post). Our interpretation of the C++11 spec on Unions (Section 9.5, page 220), is that this is undefined behavior.**

b) ****** (7 min.) Zig is an up-and-coming language that in some ways, is a direct competitor to C and C++. Let's examine a similar union in Zig.

```
const WeirdNumber = union {
  n: i64,
  f: f64,
};

test "simple union" {
  var result = WeirdNumber { .n = 123 };
  result.f = 12.3;
}
```

```
test "simple union"...access of inactive union field
.\tests.zig:342:12: 0x7ff62c89244a in test "simple union" (test.obj)
result.float = 12.3;
```

Assuming you haven't used Zig (but, crucially – you do know how to read error messages): what does this tell you about Zig's type system? How would you compare it to C++'s – and in particular, do you think one is better than the other?

First, interpreting the error: it's indicative that Zig does not allow arbitrary access for any of the types that are part of the union. Instead, we can only use the "active" union field (in Zig's language). We can verify this in the Zig docs.

In programming language terminology, this is called a tagged union.

This leans towards a more strongly-typed type system. To compare and contrast this to C++:

- **Zig's approach is "safer": it requires us to explicitly tell the language when we want to access various union fields, and prevents undefined behavior. The idea is to prevent mistakes!**
- **C++'s approach is faster. There is some overhead when Zig checks union access; C++ does not have that overhead.**